Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский физико-технический институт
(государственный университет)»

Факультет управления и прикладной математики

Кафедра теоретической и прикладной информатики

РЕАЛИЗАЦИЯ EGL И GRALLOC ДЛЯ ПАРАВИРТУАЛИЗАЦИИ GL ES ДЛЯ ANDROID.

Выпускная квалификационная работа
(бакалаврская работа)

Направление подготовки: 03.03.01 Прикладные математика и физика

Выполнил:
студент 376 группы _____ Иванов Алексей Юрьевич

Научный руководитель: _____ Корякин Алексей Борисович

г. Москва
2017

Содержание

Введение

Данная работа касается разработки компонента виртуальной машины, позволяющего гостевой системе Android использовать аппаратное ускорение графики.

Аппаратное ускорение являющееся неотъемлемой частью современных мобильных систем, так как позволяет освободить процессор от дорогостоящих графических операций. В связи неполнотой документации, сложностью реализации и большим оверхедом реализация аппаратного ускорения графики гостевой системы через виртуализацию GPU бесперспективна. По этой причине было решено паравиртуализировать OpenGL ES.

Для этого были реализованы гостевые библиотеки OpenGL ES и EGL, модуль gralloc, а также часть эмулятора, обеспечивающая их работу. Так как в процессе требовалось написание большого количества однотипного кода, был разработан генератор, выполняющий эту задачу.

Данное решение может быть использовано для запуска и отладки приложений, интенсивно использующих OpenGL ES.

Преимуществами над Google Android Emulator являются более полная поддержка стандарта OpenGL ES 2.0, а также возможность использования на устройствах, не поддерживающих настольный OpenGL, в том числе и на устройствах под управлением Android.

Работа над ускорением велась с другим студентом – Никитенко Евгением Ивановым. Каждый работал над своей частью, в частности, Евгений занимался реализацией OpenGL ES и генератором кода.

Постановка задачи

Для паравиртуализации GPU необходимо реализовать гостевые библиотеки EGL, OpenGL ES версий 1.1 и 2.0, модуль gralloc, компонент эмулятора, обеспечивающий их работу, а также механизм, выполняющий пересылку команд из гостевого кода в эмулятор.

Графическая подсистема Android

Перед тем, как переходить к реализации, следует разобрать, как работает графическая подсистема Android, а именно самая низкоуровневая ее часть, непосредственно взаимодействующая с EGL.

OpenGL ES

OpenGL – спецификация, определяющая кроссплатформенный интерфейс для написания приложений, использующих двумерную и трехмерную компьютерную графику. OpenGL ES (GLES) – его адаптация для мобильных платформ.

OpenGL ES предоставляет API для рендеринга и не дает средств для взаимодействия с оконной системой. Все взаимодействие с операционной системой вынесено в отдельную библиотеку – EGL.

В данной работе рассматриваются две версии спецификации OpenGL ES – версий 1.1 и 2.0, не совместимые между собой.

EGL

EGL предоставляет почти идентичный API для различных платформ. Прежде чем начать использовать OpenGL ES необходимо создать контекст и поверхность для рисования. Все операции GLES относятся к текущему контексту потока.

EGL определяет три вида поверхностей для рисования:

- PBuffer Surface – рисование ведется в буфер, данные которого доступны данному процессу средствами EGL. Результат может, например, использоваться в качестве текстуры.
- Window Surface – окно, отображаемое операционной системой. В случае Android рисование ведется в графический буфер, который также доступен через gralloc и может быть использован другим процессом с помощью специального расширения.
- Pixmap Surface – рисование ведется в графический буфер, реализация которого зависит от операционной системы. В случае Android не поддерживается.

OpenGL ES и EGL – лишь спецификации, и конкретные реализации зависят от оборудования, так как должны использовать его возможности.

SurfaceFlinger

На экране устройства может одновременно находиться сразу несколько окон. Так, например, статус бар является отдельным окном (в терминологии SurfaceFlinger вместо окна используются слои, но это внесло бы дополнительную путаницу при переходе к EGL).

. При этом как правило за каждое из этих окон принадлежит отдельному приложению или сервису. Перед Android стоит задача отобразить каждое из этих окон на экран.

Приложение отрисовывает содержимое окна в специальный графический буфер, и затем через BufferQueue отправляет его специальному сервису – SurfaceFlinger, который затем отображает буферы каждого из окон во фреймбуфер, выводящийся затем на экран.

Для этого SurfaceFlinger использует библиотеки OpenGL ES и EGL, при этом никак не касаясь специфики конкретного устройства. Иногда вместо них может использоваться HardwareComposer, но этот случай нас не интересует.

SurfaceView and GLSurfaceView

Фреймворк пользовательского интерфейса Android сильно зависит от иерархии его элементов (View). Для вычисления их положения используется сложный механизм, учитывающий множество факторов. Когда окно приложения становится видным, все View отрисовываются в поверхность, которую WindowManager создал для окна приложения.

SurfaceView ведет себя как обычный View до тех пор, пока дело не доходит до рендеринга: в этом случае он ведет себя, будто является абсолютно прозрачным.

Как только SurfaceView появляется на экране, SurfaceFlinger создает отдельное окно и помещает его под окно приложения (при этом прозрачность окна приложения в том месте, где располагается SurfaceView должна сделать его видимым). В дальнейшем его буферы сразу обрабатывается SurfaceFlinger и попадают на экран, избегая дополнительного копирования.

Это дает возможность отрисовывать SurfaceView в потоке, отличном от потока пользовательского интерфейса. По сути создается новое окно, положение которого тем не менее задается так же, как для элементов пользовательского интерфейса главного окна.

GLSurfaceView – обертка над SurfaceView, предоставляющая функционал GLES, беря на себя все операции EGL.

При этом на уровне EGL SurfaceView никак не отличим от настоящего окна (класс Surface). В дальнейшем не будем делать между ними различий.

BufferQueue

Класс BufferQueue отвечает связывает сущность, генерирующую графические буферы (производитель) с тем, к кто далее эти буферы использует для отображения на экран или дальнейшей обработки (потребитель). При этом производитель и потребитель могут находиться в разных процессах.

Почти вся передача графических буферов в Android происходит с помощью BufferQueue. Производитель запрашивает буфер, указывая требуемые параметры, включающие ширину, высоту и формат пикселя и флаги использования (dequeueBuffer).

В случае, когда буферы еще не были выделены или когда сменился запрашиваемый формат буфера, BufferQueue выделяет буферы с помощью gralloc.

Завершив работу с буфером, производитель возвращает его в очередь (queueBuffer).

Далее потребитель получает буфер (acquireBuffer) и, обработав его содержимое, кладет его обратно в очередь (releaseBuffer). Схема работы BufferQueue изображена на рис. 1

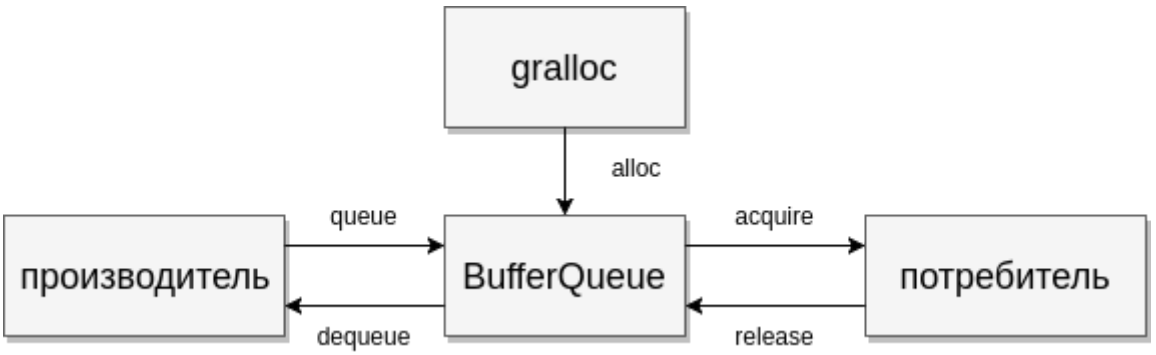

Рис. 1. Процесс работы BufferQueue

При изменении требуемых размеров буфера или его формата, запрошенных производителем или потребителем BufferQueue должен выделить новые буферы и уничтожить старые.

Window Surface

Теперь можно рассмотреть, что представляет собой Window Surface. Она создается вызовом `eglCreateWindowSurface`, принимающим ANativeWindow.

В свою очередь в ANativeWindow имеется экземпляр BufferQueue, потребителем которого как правило является SurfaceFlinger. С этого момента поверхность выступает в качестве производителя буферов и рендеринг в нее на самом деле является рендерингом в буфер, полученный из BufferQueue. Отправка отрисованного буфера происходит при вызове eglSwapBuffers.

Вся ответственность за содержимое кадровых буферов и работу с очередью ложится на EGL.

Gralloc

За создание и управление графическими буферами ответственен модуль gralloc. Так как их способ и формат хранения могут отличаться на разных устройствах, ее реализация ложится на плечи поставщика оборудования. Обычно gralloc напрямую используется для получения изображения с камеры, либо отрисовке окна без помощи OpenGL ES.

Он должен реализовывать следующие операции над буфером:

- Создание буфера заданного формата (например, RGBA8888) и размера
- Lock – отобразить буфер в оперативную память
- Unlock – убрать маппинг и применить внесенные изменения
- Register – инициализация дескриптора буфера в процессе, куда он был передан
- Unregister – уничтожение этого дескриптора
- Удаление – удаление самого буфера, вызывается только его создателем

При создании буфера gralloc передаются флаги, говорящие о его наиболее вероятном использовании и в теории способные предоставить возможность более оптимально хранить его. К сожалению, эти флаги по большей части не определяют сценарий использования буфера, а носят скорее рекомендательный характер.

Также gralloc ответственен за устройство фреймбуфера, по сути реализующее единственную значимую операцию – передать буфер для отображения на экран. Именно так полученное SurfaceFlinger финальное изображение оказывается на экране.

Расширения EGL_KHR_image и GL_OES_EGL_image_external

Графические буферы Android не являются частью стандарта OpenGL ES. Поэтому для использования такого буфера в качестве текстуры SurfaceFlinger необходимо расширить функционал библиотеки. Для этого предусмотрен механизм расширений – конкретная реализация может предоставлять дополнительные функции или расширять возможности уже существующих.

Так, расширение EGL_KHR_image вводит новый примитив – EGLImage. По сути это некоторый объект, которому поставлен в соответствие графический буфер Android. Также к нему можно “присоединять” текстуры OpenGL ES с помощью функции `glEGLImageTargetTexture2DOES`. В этом случае гарантируется, что пока буфер не захвачен с помощью операций lock/unlock, привязанные текстуры идентичны содержимому буфера.

Таким образом SurfaceFlinger получает возможность использовать содержание других окон в качестве текстур при составлении финального изображения, которое пользователь увидит на экране.

Стоит отметить, что сам буфер не обязательно был получен средствами OpenGL ES. Например, курсор в режиме отладки – буфер, содержимое которого получено с помощью lock/unlock. Таким образом курсор принадлежит отдельному сервису, но при этом отпадает необходимость создавать лишние текстуру и окно.

Жизненный цикл фреймбуфера Window Surface

Использование графического буфера, используемого в качестве кадрового буфера Window Surface, не использующегося с устройством фреймбуфера:

1. Буфер достается из очереди при создании поверхности или вызове `eglSwapBuffers` и подготавливаются для рисования.
2. OpenGL ES рендерит содержимое буфера.
3. Буфер снова попадает в очередь при вызове `eglSwapBuffers`.
4. SurfaceFlinger получает буфер из очереди, создает EGLImage, привязывает его к текстуре.
5. Текстура, привязанная к буферу используется для композиции содержимого экрана до появления в очереди нового буфера.
6. Буфер снова попадает в очередь и цикл замыкается.

В случае Window Surface, используемого с устройством фреймбуфера, буфер требуется сразу отобразить на экран и он передается в gralloc, реализующий устройство фреймбуфера.

Решение задачи

Для паравиртуализации графики нужно реализовать библиотеки EGL и OpenGL ES, а также модуль gralloc для гостевой операционной системы и часть эмулятора, с которой они будут взаимодействовать, таким образом, чтобы весь рендеринг выполнялся средствами аналогичных библиотек хоста.

Структура проекта

Для наглядности рассмотрим структа проекта:

- Гостевая часть
    - Библиотека libguest – вспомогательный функционал, нужный на госте
    - Модуль gralloc
    - Библиотека GLES
        - EGL
        - OpenGL ES 2.0
        - OpenGL ES 1.1
- Хостовая часть – реализация команд, посылаемых с гостя
- Генератор кода – генерация переброски функций с гостя, а также другого однотипного кода

Используемые библиотеки

Поскольку целевыми платформами хоста являются Linux и Android, для этого придется использовать те же EGL и OpenGL ES. В случае Linux эти библиотеки крайне ограничено поддерживаются, так что вместо них целесообразно было бы использовать GLX и OpenGL. Для этого используется ANGLE – по сути EGL и OpenGL ES, 2.0, реализованные через имеющиеся на хостовой операционной системе графические библиотеки.

Таким образом мы можем не заботиться о том, что нужно поддерживать два API.

Поверхности для рисования

В случае PBuffer интеграции с гостевой ОС не требуется, и поэтому он тривиально реализуется через PBuffer на хосте. Pixmap-surface в Android не

поддерживается, так что реализовывать ее не требуется. В случае же окна придется реализовать графический буфер, хранящий изображение на хосте.

Возможные способы реализации графического буфера

Нужно реализовать оконную поверхность EGL и, соответственно, графический буфер так, чтобы буфер, отрисованный в одном гостевом процессе, можно было бы использовать как текстуру в другом. При этом стоит заметить, что всем гостевым процессам соответствует один хостовый. Таким образом нам не нужно беспокоиться об использовании буфера в другом процессе, благодаря чему у нас появляется ряд новых возможностей помимо использования EGL_KHR_image. Итак, возможные варианты:

- Pbuffer, `glReadPixels` и `glTexImage2D`. Данный способ подразумевает копирование графического буфера при каждом использовании текстуры в оперативную память и обратно. Это, пожалуй, самый неэффективный из перечисленных способов. При этом он может работать со стандартным gralloc.
- EGL_KHR_image. К сожалению, это расширение поддерживаются далеко не идеально даже на Android. К тому же, оно не реализовано в ANGLE. К тому же необходимый API различается под Android, X11 (Linux) и Wayland (Linux).
- EGL предоставляет функцию `eglBindTexImage`, дающую использовать поверхность EGL в качестве текстуры в OpenGL ES. К сожалению, ряд реализаций EGL не дают такой возможности.
- Вместо поверхностей использовать framebuffer object (fbo), по сути представляющий собой ту же поверхность, но являющийся частью стандарта OpenGL.

Всвязи с ограниченностью поддержки функционала, используемого первыми вторым и третьим способами, обнаруженной, к сожалению, лишь после их реализации, был выбран последний способ. Однако мы все перечисленные способы.

Реализация графического буфера с помощью glReadPixels

Стандартный gralloc реализует графические буферы в виде участков разделяемой памяти, в которой хранятся значения каждого пикселя. Таким образом после отрисовки окна в Pbuffer, который в данном случае привязан к поверхности, а не к буферу результат копируется в участок разделяемой памяти с помощью `glReadPixels`.

Далее при создании EGLImage запоминается дескриптор буфера и при каждом использовании привязанной текстуры изображение загружается в текстуру вызовом `glTexImage2D`.

Очевидно, копирование графического буфера в оперативную память и обратно при каждой новой перерисовке дает большой оверхед. Однако, поскольку сам рендеринг происходит на хосте с использованием аппаратного ускорения он может давать небольшой прирост производительности.

Главным его преимуществом является необходимость специальной обработки крайне малого количества функций EGL и GLES, что делает его крайне простым в реализации.

Этот способ использовался на начальных этапах для отладки и давал небольшой прирост производительности в пользовательском интерфейсе и сравнимый с другими способами при рендеринге сложных сцен.

Реализация графического буфера с помощью EGL_KHR_image в X11

В X11 EGL_KHR_image создается из специально объекта – Pixmap. По сути это графический буфер X Server. EGL предоставляет возможность рендеринга в такой буфер с помощью Pixmap Surface.

Таким образом для каждого гостевого буфера можно создать Pixmap Surface и выполнять рендеринг в нее, а затем использовать ее для создания EGLImage хоста, ставя его в соответствие гостевому.

Для того, чтобы отобразить такой буфер в память, нужно создать XImage, представляющий собой буфер на стороне клиента с таким же форматом, как и у Pixmap, отобразить в него Pixmap и лишь затем скопировать его в гостевой буфер, преобразовывая его к правильному формату. Это необходимо из-за того что X11 как правило представляет пикселы в своем формате и при создании Pixmap можно задать лишь глубину.

Также X11 предусматривает возможность копирования региона Pixmap на другой Pixmap или Window, что планировалось использовать для реализации HardwareComposer, потенциально имеющей преимущество на GLES при отрисовке окон на экран. Однако несмотря на заявленную в документации возможность на практике это оказалось невозможным.

При тестировании было обнаружено, что в ряде реализаций этот способ работает неожиданным образом: например, перестает работать, как только было было выделено более одного буфера или неявно конвертирует Pixmap в текстуру при каждом использовании, выполняя серию дополнительных копирований.

К тому же, данное расширение не подходило для систем, поддерживающих лишь настольный OpenGL, поскольку он, как и ANGLE не предоставлял расширения EGL_KHR_image.

Реализация графического буфера с помощью eglBindTexImage

Привлекательность данного способа состояла в том, что для него не требовалось дополнительного взаимодействия с ОС: всё необходимое уже предоставлено стандартом EGL.

В этом случае графический буфер представлен PBuffer – стандартной поверхностью EGL. С помощью функции `eglBindTexImage` можно использовать его содержимое в виде текстуры и на первый взгляд реализация EGL_KHR_image кажется тривиальной. Но при этом возникают две проблемы:

- Спецификация EGL не дает никаких гарантий того, что `eglBindTexImage` будет работать для поверхности. То есть проверить это можно только попробовав использовать ее в качестве текстуры. К тому же, не гарантируется, что существуют поверхности, для которых эта операция определена. Опытным путем было выяснено, что их действительно может не быть.
- Координатные системы PBuffer и EGLImage отличаются направление оси ординат.

Если первая проблема затрагивает лишь малое количество реализация, то со второй дело обстоит гораздо сложнее. Ее можно решить двумя способами: либо дополнительным копированием с переворотом изображения, либо изменением всех текстурных координат при рендеринге в OpenGL ES.

При этом изменять координаты можно на двух этапах.

Первый вариант – переворачивать координаты во время использования такой текстуры. В случае OpenGL ES 1.1 это означает прохождение по всем вершинам в массиве, передаваемом на видеокарту, в случае OpenGL ES 2.0 – отражение координаты внутри шейдера. Проблема в том, что внутри шейдера невозможно узнать, что текстура была получена с помощью `eglBindTexImage`. Теоретически можно было бы парсить и изменять каждый шейдер, но это крайне сложная задача.

Второй вариант – отражать вершины на этапе рисования в PBuffer, но это бы изменило ориентацию полигонов и в итоге потребовало бы не меньших усилий для корректной работы в некоторых случаях. К тому же, это также требовало бы парсинга шейдера, хоть и не такого сложного.

Также можно было бы поддерживать расширение только для OpenGL ES 1.1, что хватило бы для SurfaceFlinger. К сожалению, некоторые приложения, использующие OpenGL ES 2.0 не стали бы работать, что не так критично, так как подобное имеет место в некоторых реализациях EGL на реальных устройствах.

Всвязи с перечисленными недостатками было решено, что данный способ не подходит для реализации графического буфера.

Реализация графического буфера с помощью fbo

Так как текстуры, представляющие содержимое графического буфера должны быть доступны как производителю, так и потребителю, они должны быть видны в обоих контекстах. Поскольку при создании контекстов невозможно предвидеть, для чего они будут использоваться требуется видимость всех объектов GLES во всех контекстах.

Для этого при создании нового контекста на госте соответствующий контекст на хосте разделяет все ресурсы (текстуры, шейдеры и т.п.) с главным контекстом, использующимся gralloc для работы с буферами. Это приводит к двум проблемам:

- На уровне гостевой системы не должно быть известно, что ресурсы всех контекстов разделены, что приводит к необходимости контроля их принадлежности.
- Ряд реализаций не поддерживает разделения ресурсов между контекстами OpenGL ES разных версий.

Первая проблема решается контролем над созданием и использованием ресурсов на стороне гостевой библиотеки. Все создаваемые объекты добавляются в специальную таблицу, ставящую в соответствие id гостевому объекту объект на хосте. Таким образом решаются проблемы контроля доступа и освобождения ресурсов при удалении контекста: так как ресурсы всех контекстов общие удаление контекста не приводит к освобождению созданных в нем объектов, ведь они могут использоваться в других контекстах.

Вторая проблема значительнее сложнее. Фактически мы вынуждены использовать единственную версию GLES, чтобы поддерживать такие реализации. Таким образом, возникает необходимость реализовать GLES1 средствами GLES2, в котором была прекращена поддержка ряда функций.

Это также делает возможным использование ANGLE, которая не реализует OpenGL ES версии 1.1.

Так как графический буфер не всегда рендерится средствами OpenGL ES и даже не всегда используется в качестве текстуры, а отображение графического буфера в память и обратно – довольно дорогостоящие операции, в случае, когда буфер не используется в качестве цели при рендеринге, он будет представлен участком разделяемой памяти (он должен

быть доступен множеству процессов). К тому же, буферы формата YUV не поддерживаются стандартом EGL, что исключает возможность их хранения в виде текстуры или fbo + текстуры. Создание текстуры для рисования таких буферов выполняется только при создании соответствующего EGLImage, соответствующего этому буферу. Таким образом на самом деле буфер может быть одним из следующих способов:

- Разделяемая память (ashmem) – буферы, не используемые GLES каким-либо образом.
- Разделяемая память + текстура – буферы, используемые в качестве текстуры, но не являющиеся целью рендеринга. Разделяемая память остается для ускорения отображения в оперативную память и облегчения синхронизации.
- Текстура + fbo – буферы, в которые выполняется рендеринг. Разделяемая память используется только при lock на чтение.

Если к буферу привязан EGLImage, все изменения, внесенные в буфер с помощью gralloc (lock/unlock) должны немедленно быть применены к текстуре. Для этого содержимое разделяемой памяти загружается в текстуру с помощью вызова glTexImage2D. Таким образом если у буфера нет текстуры, она создается, а если она уже есть – обновляется, Так как при привязанном fbo это сделать невозможно, приходится уничтожать fbo и создавать текстуру заново. К счастью, такой сценарий использования буфер крайне маловероятен и пересоздание текстуры не привносит большого оверхеда.

В случае, когда буфер все-таки нужно отрисовать средствами OpenGL ES, создается специальный объект – fbo, по функционалу идентичный поверхности EGL. При этом текстура указывается в качестве так называемого буфера цвета – буфера, в котором окажется итоговое изображение. Также необходимо создать буфер глубины – буфер, который

отвечает за то, чтобы ближние объекты отличались от дальних и перекрывали их в итоговом изображении. В случае поверхностей такой буфер создается неявно. Далее для рисования вместо оконной поверхности для рисования используется fbo ее текущего буфера. Возможные состояния графического буфера и переходы между ними изображены на рис. 2.

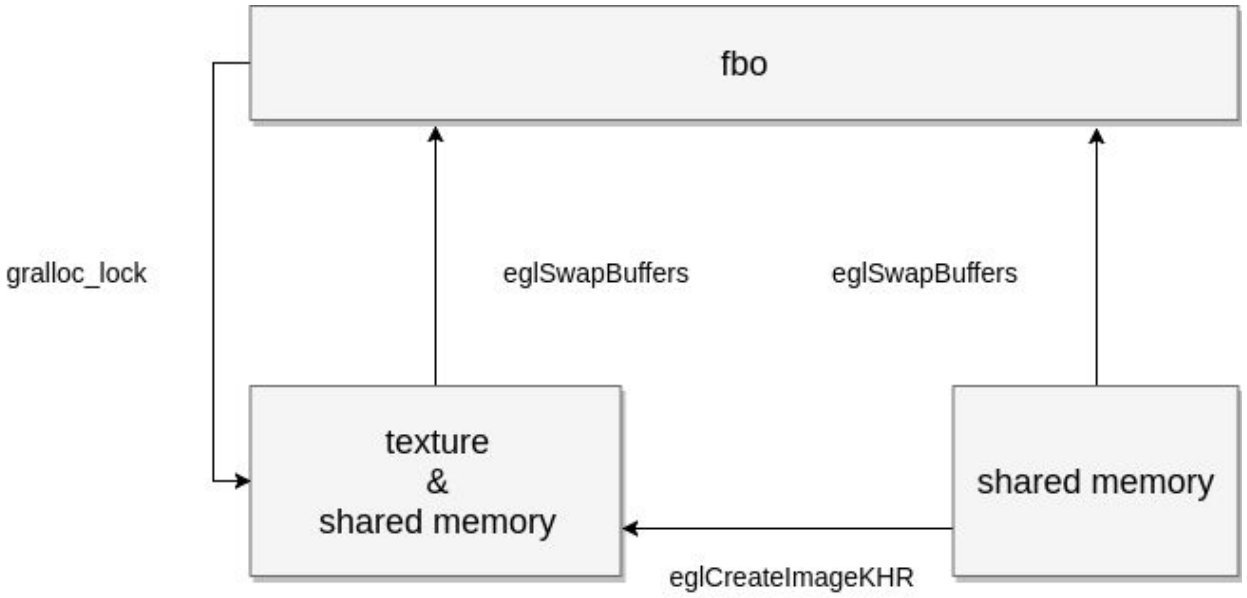

Рис. 2. Возможные состояния графического буфера

При этом отдельно нужно обработать случай, когда само пользовательское приложение использует fbo. В этом случае рендеринг производится не в поверхность и подменять цель рисования не требуется.

Таким образом после отрисовки буфера у нас уже есть готовая текстура, и реализация EGLImage оказывается тривиальной.

Передача команд с гостя на хост

Для взаимодействия гостевых библиотек с эмулятором нужно иметь возможность исполнять команды с гостя на хосте. Это было реализовано

следующим образом: параметры функции записываются в буфер, с помощью инструкции BXJ, не используемой в Android. После распаковки параметров вызывается соответствующая хостовая функция. При этом весь код, пакующий и распаковывающий параметры генерируется автоматически, и со стороны это выглядит как обычный вызов функции.

Так как переключение управления на хост – дорогостоящая с точки производительности операция, нужно минимизировать частоту передачи управление. При этом есть три случая, когда это необходимо сделать:

- Функция возвращает значение или выполняет запись в буфер
- Функция должна выполниться сразу (`eglSwapBuffers`, `glFinish`)
- Буфер команд заполнен

Таким образом последовательность команд записывается в буфер данного потока, который исполняется и очищается только при необходимости. Эта оптимизация особенно хорошо работает для функций, непосредственно выполняющих рендеринг, так как они не возвращают значений и могут не исполняться до вызова eglSwapBuffers.

Так как в случае GLES у каждого потока есть свои текущие контекст и поверхность, а всем процессам и потокам гостя соответствует всего несколько потоков хоста, необходимо выставить текущий контекст при каждом исполнении буфера. Для этого в начало буфера вставляется пролог, ответственный за их выставление.

При этом данный механизм используется как gralloc, так и GLES, и вызовы их функций на хосте должны происходить в том же порядке, что и на госте, то есть на обе библиотеки стоит иметь единственный буфер. По этой причине гостевая часть реализации буфера команд выполнена в отдельной библиотеке – libguest.

Генератор кода

Переброска каждой функции с гостя на хост требует сериализации ее вызова в буфер команд, а затем десериализации на стороне хоста. Так как таких функций довольно много, целесообразно генерировать код, занимающийся этим автоматически.

Для этого была написана специальная утилита. Так как в функцию могут передаваться буферы переменного размера как на чтение, так и на запись, а некоторые должны быть вызваны на хосте немедленно. Есть еще ряд специальных случаев, каждый из которых необходимо правильно обработать.

Для этого все перебрасываемые функции имеют описания – хинты, использующиеся генератором для их правильной обработки. Так как требуемое поведение генератора является довольно сложным и порой не совсем очевидным, а результат его работы использовался несколькими разработчиками, была написана спецификация, в которой четко прописаны правила обработки хинтов. Данную спецификацию можно найти в Приложении А.

Также генератор кода ответственен за ряд других задач, требующих написания однотипного кода.

Гостевая часть EGL

Основные функции гостевой части EGL:

- Инициализация контекстов OpenGL ES и разделение их ресурсов (на уровне гостя) при необходимости
- Управление состоянием потока (текущие контекст и поверхность, ошибки EGL), его синхронизация при передаче управления на хост

- Работа с графическими буферами для оконной поверхности, общение с BufferQueue
- Управление ресурсами и подсчет ссылок – в EGL объект удаляется только после запроса на его удаление и обнуления количества ссылок на него
- Создание EGLImage, реализующееся тривиально, поскольку рендеринг уже происходит в текстуру

Хостовая часть EGL

Большинство дескрипторов объектов, создаваемых EGL (например, контекстов и поверхностей) являются указателями, которые как правило без каких-либо проверок разыменовываются внутри библиотеки. По этой причине нельзя принимать дескрипторы, переданные с гостя, без проверки из соображений безопасности.

Для этого они заносятся в специальные таблицы, соответствующие каждому процессу. Разделение по процессам добавлено не столько для их изоляции, сколько для освобождения ресурсов хоста, по какой-либо причине не освобожденных гостевым процессом.

Также хостовая часть EGL ответственна за выделение хостовых контекстов, ставящихся в соответствие гостевым, и поиск фреймбуферных конфигураций, удовлетворяющих заданным требованиям. Это не такая тривиальная задача, как может показаться на первый взгляд, так как спецификация EGL не говорит, при каких условиях контексты могут разделять ресурсы. Поэтому приходится проверять конфигурации на совместимость с заранее выбранной основной и, если получен отрицательный результат, скрывать ее существование от гостя.

Поскольку за текущие контекст и поверхность отвечает EGL, выставление в качестве текущего fbo графического буфера и его инициализация в случае рендеринга в гостевую оконную поверхность также выполняется им.

Так как содержимое экрана также отрисовывается с помощью EGL, можно обрабатывать соответствующую поверхность отдельно, вместо того, чтобы сначала рисовать в кадровый буфер, выполнять рендеринг непосредственно в окно эмулятора. В таком случае создается окно, совместимое с конфигурацией SurfaceFlinger.

Результаты

В результате работы был разработан компонент эмулятора, а также соответствующие гостевые библиотеки, позволяющие значительно увеличить производительность графики на гостевой системе. Особенно это заметно в трехмерных приложениях.

Система, а также тесты glmark2 (приложение, измеряющее производительность OpenGL ES 2.0) работают корректно. В общей сложности код, за вычетом автоматически генерируемого, занимает более 21000 строк. Скриншоты работающего эмулятора можно наблюдать на рис. 3 и рис. 4.

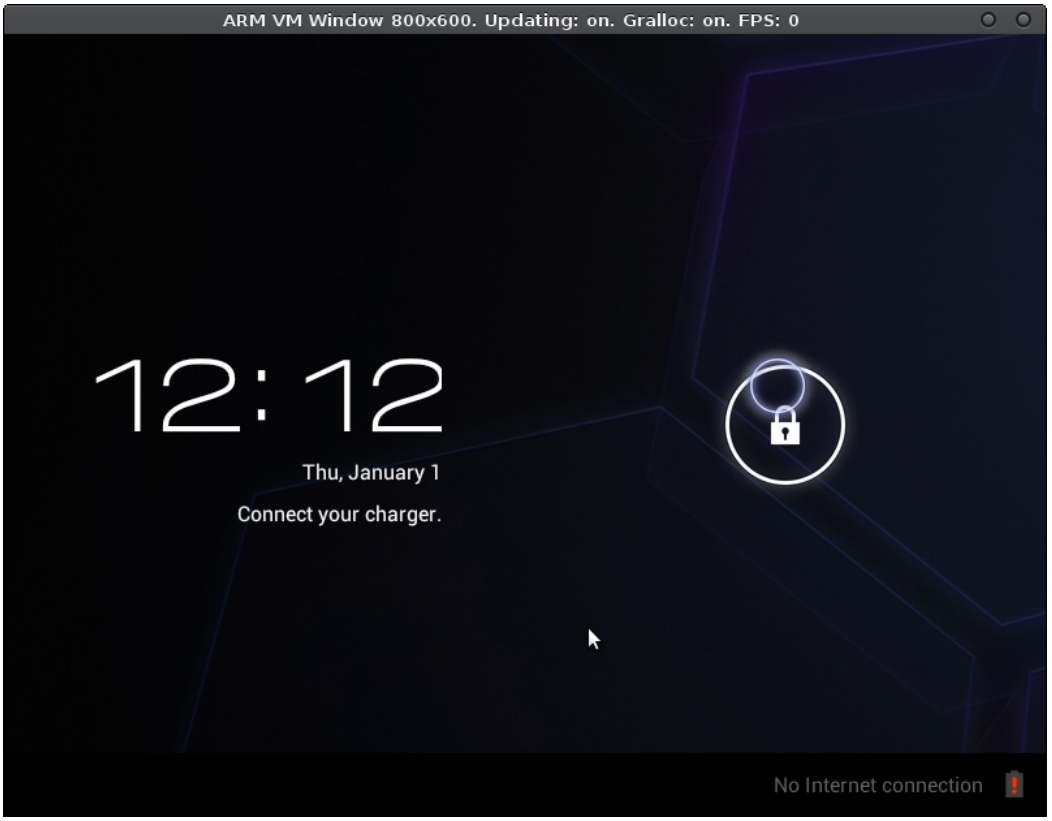

Рис. 3. Результат работы эмулятора. Экран блокировки.

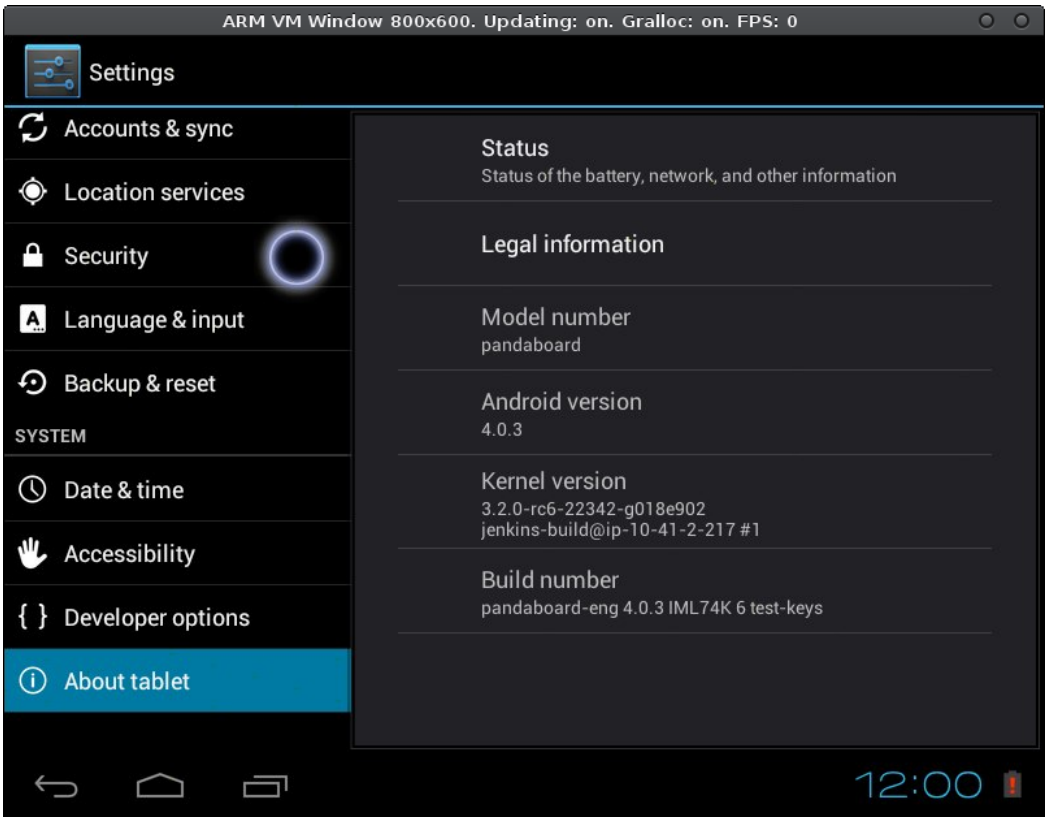

Рис. 4. Результат работы эмулятора. Настройки Android.

Выводы

Данная работа имеет два может иметь ряд преимуществ над реализацией ускорения графики в Google Android Emulator, который является наиболее популярным эмулятором Android. Благодаря использованию библиотеки ANGLE, реализующей OpenGL ES 2.0 средствами OpenGL, почти полностью поддерживается стандарт OpenGL ES 2.0, а также возможна работа на устройствах, не поддерживающих настольный OpenGL.

Благодаря полноте поддержки стандарта эмулятор вполне можно использовать для отладки приложений, использующих OpenGL ES 2.0. К тому же, даже в стандартных приложениях скорость работы Android под эмулятором значительно повышается.

Так как большая часть функционала реализовано на госте, а графические буферы реализованы стандартными средствами OpenGL ES 2.0, адаптация части эмулятора, ответственной за ускорение графики, должна быть довольно проста. В дальнейшем планируется ее использование в Parallels Desktop.

Список источников

1. Implementing graphics – https://source.android.com/devices/graphics/implement
2. The Android graphics path – http://elinux.org/images/2/2b/Android_graphics_path--chis_simmonds.pdf
3. EGL reference pages – https://www.khronos.org/registry/EGL/sdk/docs/man/
4. EGL 1.5 specification – https://www.khronos.org/registry/EGL/specs/eglspec.1.4.pdf
5. EGL_KHR_image specification – https://www.khronos.org/registry/EGL/extensions/KHR/EGL_KHR_image.txt
6. EGL_ANDROID_image_native_buffer specification – https://www.khronos.org/registry/EGL/extensions/ANDROID/EGL_ANDROID_image_native_buffer.txt
7. OES_EGL_image_external specification – https://www.khronos.org/registry/OpenGL/extensions/OES/OES_EGL_image_external.txt
8. Charles blog. Android – https://charleszblog.wordpress.com/tag/android/
9. OpenGL and OpenGL ES reference pages – https://github.com/KhronosGroup/OpenGL-Refpages

10. ANGLE repository – https://chromium.googlesource.com/angle/angle/+/master/
11. Mozilla wiki. Gralloc – https://wiki.mozilla.org/Platform/GFX/Gralloc
12. *John Kessenich, Graham Sellers, Dave Shreiner* – OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V (9th Edition)
13. OpenGL ES Common Profile Specification Version 2.0.25 (Full Specification) (November 2, 2010) – https://www.khronos.org/registry/OpenGL/specs/es/2.0/es_full_spec_2.0.pdf

Приложение А. Формат GLHL-файлов

Далее будет описан формат GLHL-файла с дополнительным описанием функций, которое необходимо для сериализации и десериализации функций.

GLHL (OPENGL HINTS LANGUAGE)

Version 1.8

Date: 11.04.2017

Это спецификация стандарта GLHL

1. Общие положения

1.1. Задача – имея заголовочный файл OpenGL функций, сгенерировать код пересылки функций на host либо же специальным образом реализовать функции и другие вещи для реализации OpenGL на guest. Генератор кода занимается этой задачей. На вход ему подаются файлы с функциями для транслирования и специальные файлы (далее – glhl-файлы), в которых описывается, как нужно обрабатывать функцию. На выходе – сгенерированные файлы для host и guest.

1.2. В этом документе будет описан стандарт glhl.

1.3. Гарантируется, что размеры всех типов, которые используются, кроме указателей, одинаковы на host и guest, форматы типов с плавающей точкой совпадают и packing структур, которые передаются, одинаков на host и guest.

1.4. Каждый заголовочный файл относится к библиотеке на guest, в параметрах генератора кода можно указать принадлежность файла к библиотеке.

1.5. Список допустимых библиотек: gralloc, guest, egl, gl.

1.6. Для функций из библиотеки gl будет выбираться нужная, в зависимости от текущего контекста.

2. Формат glhl-файла

2.1. У всех файлов формата GLHL рекомендуется добавлять расширение glhl.

2.2. Все glhl-файлы, в которых есть функции, должны начинаться с имени библиотеки.

2.3. Комментарий – `# комментарий`. Может быть в любой части строки. Все, что идет после него на этой строке, игнорируется.

2.4. Любые пустые строки игнорируются (включая строки с пробелами, табуляциями и комментариями).

2.5. Все лишние пробелы и табуляции допустимы (не считая исключения в пунктах 6.4 и 6.5).

3. `@include "путь.glhl"` – подключение файла

3.1. Подключения файлов могут быть в любом месте. Вместо подключения файла идет содержимое другого файла.

3.2. Циклическое подключение файлов запрещено.

3.3. Необходимо указывать путь относительно исходного файла.

3.4. Абсолютные пути (которые начинаются со слеша) запрещены.

4. Хинты

4.1. В файле хранятся функции, параметры и их хинты – указание на то, как должны обрабатываться функции и параметры.

4.2. У функций и параметров может быть несколько хинтов.

4.3. У параметров нет стандартного хинта. Если необходимо стандартное поведение, то для данной функции не нужно указывать параметр.

4.4. Некоторые параметры не могут не иметь хинта, тогда необходимо указать хинт.

4.5. Все функции должны иметь хинты.

4.6. У хинтов параметров может быть атрибут (но только один), у функций – нет.

4.7. Порядок хинтов у функций и параметров неважен.

4.8. У функций, которые находятся в файле, должен быть хотя бы один хинт.

4.9. У параметров, которые находятся в файле, должен быть хотя бы один хинт.

4.10. Одна функция и один параметр функции не могут встречаться больше чем один раз.

4.11. Один хинт для функции или параметра не может встречаться больше чем один раз.

4.12. Если функция имеет хинт `no_implement`, `alias`, `custom_data` или `unsupported`, то она не может иметь хинтов параметров.

5. Типы

5.1. Все типы, у которых отсутствует явное указание на то, что это массив или указатель, не являются массивом или указателем.

5.2. У функции не может быть двойного массива в качестве параметра.

5.3. Если функция имеет двойной (или более) указатель или указатель (одинарный или более) на массив, то функция должна иметь хинт `no_implement`, `alias`, `custom_data` или `unsupported`.

5.4. Если какой-то параметр функции – указатель или массив, то для него обязан быть хинт (либо у функции должен быть хинт `no_implement`, `alias`, `custom_data` или `unsupported`).

5.5. Если у функции есть параметр одномерный массив, то для него должен быть указан его размер в исходном файле.

5.6. Если возвращаемое значение является указателем, то для функции должен быть указан хинт `no_implement`, `alias`, `custom_data` или `unsupported`.

5.7. С-структура, у которой явно присутствует слово `struct`, не может быть у функций в качестве параметра или возвращаемого значения в исходных файлах.

6. Синтаксис

6.1. Если не требуется специальной обработки параметров, то хинты для функций будут выглядеть так:

```
[hint1, hint2, ... hintN] имя_функции
```

6.2. Если есть параметры, для которых нужны хинты (без дополнительных параметров), то синтаксис будет таким (табуляция обязательна):

```
[hint1, hint2, ... hintN] имя_функции
    хинт_параметра_1
    хинт_параметра_2
    ...
    хинт_параметра_N
```

Где `хинт_параметра_I`:

```
[paramI_hint1, paramI_hint2, ... paramI_hintN] имя_параметра_I
```

6.3. Только у одного хинта параметра есть дополнительный параметр:

[paramI_hint1, paramI_hint2, ... paramI_hintN]
имя_параметра_I: доп_параметр

6.4. Хинты функций должны начинаться сразу с начала строки (без пробелов).

6.5. Перед хинтами параметров должна быть ровно одна табуляция.

7. Хинты функций: guest_custom, no_implement, default, no_state, unsupported.

7.1. default – поведение по умолчанию.

Совместимые хинты: нет

7.2. guest_custom – на guest имя функции будет video_имя_функции, а не имя_функции.

Совместимые хинты: flush

7.3. no_implement – для функции не выполняется никаких действий и не добавляется в таблицу video_call.

Совместимые хинты: нет

7.4. unsupported – вместо вызова функции на host печатается предупреждение с именем функции. Эта функция не будет добавляться в список eglGetProcAddress.

Совместимые хинты: нет

7.5. flush – буфер команд будет принудительно исполнен на host.

Совместимые хинты: guest_custom

7.6. custom_data – функция будет добавлена в video call table, будет сгенерирована константа – video call id, сами обертки не будут созданы на guest и host.

Совместимые хинты: нет

7.7. `alias` – только для gl. Вместо этой функции вызывать такую же, но без `OES`, `IMG`, `NV`, `KHR`, `OVR` и `QCOM` в конце. Причем если функция есть в gl2, то будет вызвана она, в противном случае будет вызвана gl1 функция.

Совместимые хинты: нет

8. Хинты параметров: `in`, `out`, `array_in`, `array_out`, `null`.

8.1. `in` – копировать данные на guest после вызова соответствующей функции.

Дополнительные параметры: размер, вычисляемый на guest

Только для: указателей

Совместимые хинты: `null`

8.2. `out` – копировать данные с host.

Дополнительные параметры: размер, вычисляемый на guest

Только для: указателей

Совместимые хинты: `null`

8.3. `array_in` – копировать массив на guest после вызова соответствующей функции.

Дополнительные параметры: нет

Только для: массивов

Совместимые хинты: нет

8.4. `array_out` – копировать массив с host.

Дополнительные параметры: нет

Только для: массивов

Совместимые хинты: нет

8.5. `null` – указатель может быть равен нулю, в таком случае данные не будут копироваться. Без этого хинта получение нулевого указателя будет являться ошибкой. С этим хинтом значения `size_in` и `size_out` размера будут вычисляться, только если указатель не `NULL`.

Дополнительные параметры: нет

Только для: указателей

Совместимые хинты: `in`, `out`